# 13 | 高性能网络实战（下）：如何完善负载均衡器？

2022-02-14 倪朋飞

《eBPF 核心技术与实战》 课程介绍 >

**讲述：山荣**
时长 15:20 大小 14.06M

你好，我是倪朋飞。

上一讲，我带你使用 sockops 和 sk_msg 等套接字 eBPF 程序，在内核态对套接字进行转发，提升了负载均衡的性能。

对于网络优化来说，除了套接字 eBPF 程序，XDP 程序和 TC 程序也可以用来优化网络的性能。特别是 XDP 程序，由于它在 Linux 内核协议栈之前就可以处理网络包，在负载均衡、防火墙等需要高性能网络的场景中已经得到大量的应用。

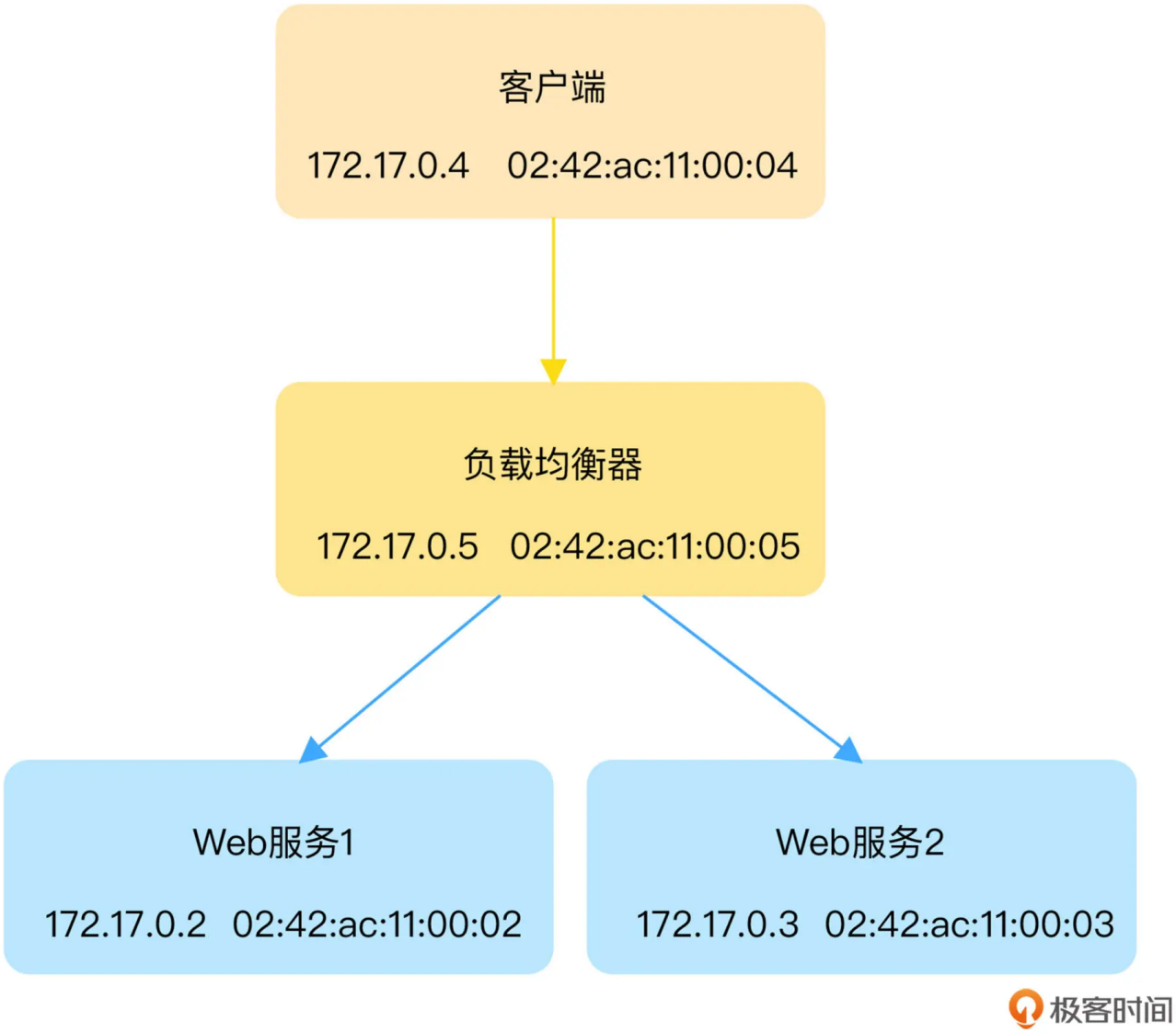
XDP 程序在内核协议栈初始化之前运行，这也就意味着在 XDP 程序中，你并不能像在 sockops 等程序中那样直接获得套接字的详细信息。使用 XDP 程序加速负载均衡，通常也就意味着需要从头开发一个负载均衡程序。这是不是说 XDP 的使用就特别复杂，需要重新

实现内核协议栈的很多逻辑呢？不要担心，**XDP 处理过的数据包还可以正常通过内核协议栈继续处理，所以你只需要在 XDP 程序中实现最核心的网络逻辑就可以了。**

今天，我就以 XDP 程序为例，带你继续优化和完善负载均衡器的性能。

## 案例准备

跟上一讲类似，为了方便环境的重现，负载均衡器、Web 服务器以及客户端都还是运行在容器中，它们的 IP 和 MAC 等基本信息如下图所示：




执行下面的命令，启动这几个容器：

复制代码

```
# Webserver
```

```
docker run -itd --name=http1 --hostname=http1 feisky/webserver
docker run -itd --name=http2 --hostname=http2 feisky/webserver

# Client
docker run -itd --name=client alpine

# LB
docker run -itd --name=lb --privileged -v /sys/kernel/debug:/sys/kernel/debug
```

> 小提示：在默认安装的 Docker 环境中，假如你没有运行其他容器，运行上述命令后得到的 IP 地址跟图中是相同的。

注意，我们把作为负载均衡器的 Nginx 换成了基于 alpine 镜像的 SHELL 容器，并且以特权容器的方式运行，以便有足够的权限加载并运行 XDP 程序。

把 XDP 程序放入容器中，除了容易复现案例环境之外，在开发和调试 XDP 程序的过程中也不会影响主机的网络（否则，错误的 XDP 程序可能导致主机网络中断，进而也会影响远程 SSH 连接）。

由于负载均衡容器只启动了一个 SHELL 环境，并没有运行真正的负载均衡服务。此时，访问负载均衡器的 TCP 80 端口会直接失败。你可以运行下面的命令到客户端容器中验证（/ # 后的命令表示在容器终端中运行）：

复制代码

```
docker exec -it client sh
/ # apk add curl --update
/ # curl "http://172.17.0.5"
curl: (7) Failed to connect to 172.17.0.5 port 80 after 1 ms: Connection refus
```

案例所需的容器环境启动完毕后，接下来我们再来看看，如何使用 XDP 开发一个负载均衡服务。由于需要把 XDP 字节码放到容器中运行，本着最小依赖的原则，我们将使用 libbpf 作为 XDP 的基础库，这样只需要把编译后的二进制文件放入容器中即可运行。

## 如何用 XDP 开发一个负载均衡器？

还记得我在 08 讲 中提到过的基于 libbpf 开发 eBPF 程序的基本步骤吗？不记得也没关系，我们再来回顾一下。 libbpf 的使用通常分为以下几步：

1. 开发 eBPF 程序，并把源文件命名为 <程序名>.bpf.c；
2. 编译 eBPF 程序为字节码，然后再调用 bpftool gen skeleton 为 eBPF 字节码生成脚手架头文件；
3. 开发用户态程序，引入生成的脚手架头文件后，加载 eBPF 程序并挂载到相应的内核事件中。

接下来，我们就按这几个步骤来开发 XDP 负载均衡程序。

## 开发 XDP eBPF 程序

第一步是开发一个运行在内核态的 eBPF 程序。参考内核中 BPF_PROG_TYPE_XDP 程序类型的定义格式，它的参数类型为 struct xdp_md：

复制代码

```
#define BPF_PROG_TYPE(_id, _name, prog_ctx_type, kern_ctx_type)

BPF_PROG_TYPE(BPF_PROG_TYPE_XDP, xdp,
        struct xdp_md, struct xdp_buff)
```

因而，你就可以使用如下的格式来定义这个 XDP 程序：

复制代码

```
SEC("xdp")
int xdp_proxy(struct xdp_md *ctx)
{
  // TODO: 添加XDP负载均衡逻辑
}
```

这段代码中，SEC("xdp") 表示程序的类型为 XDP 程序。你可以在 libbpf 中 section_defs找到所有 eBPF 程序类型对应的段名称格式。

参考 linux/bpf.h 头文件中 struct xdp_md 的定义格式，你可以发现，它比上一讲用到的 struct bpf_sock_ops 简单多了，只包含了如下的几个字段：

复制代码

```
struct xdp_md {
    __u32 data;
    __u32 data_end;
    __u32 data_meta;
    /* Below access go through struct xdp_rxq_info */
    __u32 ingress_ifindex; /* rxq->dev->ifindex */
    __u32 rx_queue_index;  /* rxq->queue_index  */
    __u32 egress_ifindex;  /* txq->dev->ifindex */
};
```

从 struct xdp_md 的定义中你可以看到，所有的字段都是整型数值，其中前三个表示数据指针信息（包括开始位置、结束位置、元数据位置），而后三个表示关联网卡的信息（包括入口网卡、入口网卡队列以及出口网卡的编号）。

由于 struct xdp_md 中并不包含 skb 数据结构，在 XDP 程序中，你只能通过 data 和 data_end 这两个指针去访问网络报文数据。而要想利用原始网络数据指针来访问网络数据，就需要你了解 TCP/IP 网络报文的基本格式。

为了方便你理解，我画了一张图，标记了以太网头、IP 头以及 TCP 头等相对于 struct xdp_md 中数据指针的位置关系：

极客时间

有了这些对应关系，要访问 TCP/IP 协议某一层的头结构，就可以使用开始指针 data 再加上它之前所有头结构大小的偏移。

比如，对于以太网头，它的位置跟开始位置 data 是相同的，因而你就可以使用下面的方式，把它转换为指针格式进行访问：

复制代码

```
void *data = (void *)(long)ctx->data;
void *data_end = (void *)(long)ctx->data_end;
```

```
3
4 struct ethhdr *eth = data;
5 if (data + sizeof(struct ethhdr) > data_end)
6 {
7   return XDP_ABORTED;
8 }
```

为了帮助 eBPF 校验器验证数据访问的合法性，在访问以太网头数据结构 struct ethhdr 之前，你需要检查数据指针是否越界。如果检查失败，就要返回一个错误（这儿返回的 XDP_ABORTED 表示丢弃数据包并记录错误行为以便排错）。

了解了太网头的访问格式之后，IP 头的访问也是类似的。在开始指针 data 之后加上太网头数据结构的长度偏移，就是 IP 头所指向的位置。拿到 IP 头之后，你还可以对网络数据进行初步的校验，比如忽略 IPv6、UDP 等我们不关心的数据，只处理 TCP 数据包，代码如下：

复制代码

```
1 struct iphdr *iph = data + sizeof(struct ethhdr);
2 if (data + sizeof(struct ethhdr) + sizeof(struct iphdr) > data_end)
3 {
4   return XDP_ABORTED;
5 }
6
7 if (eth->h_proto != bpf_htons(ETH_P_IP))
8 {
9   return XDP_PASS;
10 }
11
12 if (iph->protocol != IPPROTO_TCP)
13 {
14   return XDP_PASS;
15 }
```

这段代码中返回的 XDP_PASS 表示把网络包传递给内核协议栈，内核协议栈接收到网络包后，按正常流程继续处理。

进行了基本的校验之后，再接下来就是实现负载均衡的逻辑了。由于我们想要实现的是一个四层负载均衡，试想一下，负载均衡器收到客户端的请求之后，需要把目的地址（包括 IP 和 MAC）替换成后端 Webserver 的地址，再重新发到内核协议栈中继续处理。

参考内核中以太网头和 IP 头的定义格式， IP 地址都是以 __be16 类型的大端格式存储，而 MAC 地址则是以字符数组 unsigned char h_dest[6] 的形式存储。因而，MAC 地址可以直接通过数据下标进行访问，而我们案例开始时列出的 IP 地址，还需要先转换为 __be16 格式的大端存储格式。

如果你还不熟悉 IP 地址的转换方法，那可以参考下面的程序，调用 inet_addr() 库函数帮你完成转换：

复制代码

```
#include <stdio.h>
#include <arpa/inet.h>

int main() {
  unsigned int a1 = inet_addr("172.17.0.2");
  unsigned int a2 = inet_addr("172.17.0.3");
  unsigned int a3 = inet_addr("172.17.0.4");
  unsigned int a4 = inet_addr("172.17.0.5");
  printf("0x%x 0x%x 0x%x 0x%x\n", a1, a2, a3, a4);
}
```

为了方便你理解接下来的程序，我把转换后的容器地址信息整理成了一个表格，你可以在后续的开发和问题排查过程中参考：

| 容器名称 | IP地址(字符串格式) | IP地址(整型格式) | MAC 地址 |
| --- | --- | --- | --- |
| client | 172.17.0.4 | 0x40011ac | 02:42:ac:11:00:04 |
| lb | 172.17.0.5 | 0x50011ac | 02:42:ac:11:00:05 |
| http1 | 172.17.0.2 | 0x20011ac | 02:42:ac:11:00:02 |
| http2 | 172.17.0.3 | 0x30011ac | 02:42:ac:11:00:03 |

极客时间

接下来，就是负载均衡的实现过程了，也就是根据请求的来源，把目的地址修改为 Webserver 的地址。下面的代码展示的就是一个最简单的负载均衡实现逻辑：

复制代码

```
/* 1. 常量定义 */
#define CLIENT_IP 0x40011ac
#define LOADBALANCER_IP 0x50011ac
#define ENDPOINT1_IP 0x20011ac
#define ENDPOINT2_IP 0x30011ac
#define CLIENT_MAC_SUFFIX 0x04
#define LOADBALANCER_MAC_SUFFIX 0x05
#define ENDPOINT1_MAC_SUFFIX 0x02
#define ENDPOINT2_MAC_SUFFIX 0x03

/* 2. 从客户端发送过来的请求，目的地址改为后端 Webserver 的地址 */
if (iph->saddr == CLIENT_IP)
{
  iph->daddr = ENDPOINT1_IP;
  eth->h_dest[5] = ENDPOINT1_MAC_SUFFIX; /* Only need to update the last byte

  /* 模拟从两个Webserver随机选择 */
  if ((bpf_ktime_get_ns() & 0x1) == 0x1)
  {
    iph->daddr = ENDPOINT2_IP;
    eth->h_dest[5] = ENDPOINT2_MAC_SUFFIX;
  }
```

```
23 }
24 else /* 3. 反之，目的地址改为客户端 */
25 {
26   iph->daddr = CLIENT_IP;
27   eth->h_dest[5] = CLIENT_MAC_SUFFIX;
28 }
29
30 /* 4. 修改原地址为LB地址 */
31 iph->saddr = LOADBALANCER_IP;
32 eth->h_source[5] = LOADBALANCER_MAC_SUFFIX;
```

这段代码中各部分的含义如下：

第 1 部分，将容器地址信息定义为常量，方便后续引用和理解。注意，MAC 地址是一个包含 6 个元素的数组，而前 5 个元素的值都是相同的。因而，在更新目的 MAC 地址时，只需要更新最后一个元素即可。所以，常量定义里面也只包含了最后一个字节的值。

第 2 部分，对于从客户端发送过来的请求，将目的地址改为后端 Webserver 的地址。由于只有两个后端 Webserver，这儿使用时间戳最后一位的值模拟它们的随机选择过程。

第 3 部分，对于从 Webserver 发送过来的响应，目的地址改为客户端地址。

第 4 部分，将原地址都改为负载均衡器的地址。

到这里， eBPF 程序是不是已经开发好了呢？其实，如果你了解过 IP 协议的基本原理，你就知道还有一个步骤也是非常重要的：由于修改了 IP 头的数据，IP 头的校验和（checksum）就需要重新计算，否则网络包会被内核直接丢弃。

如果你不熟悉 IP 头校验和的计算方法也没关系，你可以很容易从成熟的开源项目中查找到相关的计算方法。比如，参考 Facebook 开源的 Katran，你可以定义如下的 `ipv4_csum()` 函数来计算校验和：

复制代码

```
1 static __always_inline __u16 csum_fold_helper(__u64 csum)
2 {
3   int i;
4 #pragma unroll
5   for (i = 0; i < 4; i++)
6   {
```

```
  if (csum >> 16)
    csum = (csum & 0xffff) + (csum >> 16);
  }
  return ~csum;
}

static __always_inline __u16 ipv4_csum(struct iphdr *iph)
{
  iph->check = 0;
  unsigned long long csum = bpf_csum_diff(0, 0, (unsigned int *)iph, sizeof(st
  return csum_fold_helper(csum);
}
```

关于校验和的具体算法，你可以参考 TCP/IP 协议相关的原理书籍（如《TCP/IP 详解》）来理解，这里我就不详细展开了。

有了校验和的计算方法之后，最后更新 IP 头的 checksum，再返回 XDP_TX 把数据包从原网卡发送出去，交给内核去转发就可以了。下面展示的就是更新校验和的实现方法：

复制代码

```
SEC("xdp")
int xdp_proxy(struct xdp_md *ctx)
{
  ...
  /* 重新计算校验和 */
  iph->check = ipv4_csum(iph);

  /* 把数据包从原网卡发送出去 */
  return XDP_TX;
}
```

把上述代码保存到一个文件 xdp-proxy.bpf.c 中，就完成了 XDP eBPF 程序的开发（你还可以在 GitHub 中找到完整的代码）。

## 编译并生成脚手架头文件

有了 XDP 程序之后，接下来的第二步就比较简单了。我们只需要执行下面的 clang 命令，把 XDP 程序编译成字节码，再执行 bpftool gen skeleton 命令生成脚手架头文件即可：

```
clang -g -O2 -target bpf -D__TARGET_ARCH_x86 -I/usr/include/x86_64-linux-gnu
bpftool gen skeleton xdp-proxy.bpf.o > xdp-proxy.skel.h
```

复制代码

## 开发用户态程序

对于第三步用户态程序的开发，它的基本流程跟 08 讲 中的内核跟踪案例是类似的，也是需要引入脚手架头文件、增大 RLIMIT_MEMLOCK、初始化并加载 BPF 字节码，最后再挂载 XDP 程序这几个步骤。忽略错误处理步骤，最核心的实现步骤如下所示：

复制代码

```
// 1. 引入脚手架头文件
#include "xdp-proxy.skel.h"

// C语言主函数
int main(int argc, char **argv)
{
    // 2. 增大 RLIMIT_MEMLOCK（默认值通常太小，不足以存入BPF映射的内容）
    struct rlimit rlim_new = {
      .rlim_cur = RLIM_INFINITY,
      .rlim_max = RLIM_INFINITY,
    };
    err = setrlimit(RLIMIT_MEMLOCK, &rlim_new);

    // 3. 初始化BPF程序
    struct xdp_proxy_bpf *obj = xdp_proxy_bpf__open();

    // 4. 加载BPF字节码
    err = xdp_proxy_bpf__load(obj);

    // 5. TODO: 挂载XDP程序到eth0网卡
}
```

这段代码中，前面 4 个步骤跟 08 讲 中的内核跟踪案例是一样的，这儿不再详细展开。

而对于第 5 步的挂载过程，我在 06 讲 中曾经提到，你可以使用 `ip link` 命令来挂载 XDP 程序。当时讲到的是在主机中挂载 XDP 的步骤，而在容器中的步骤其实也是一样的（注意，在容器中的虚拟网卡上，只支持以通用模式挂载）。

下面的代码展示的就是把 XDP 字节码复制到负载均衡容器并挂载到 eth0 网卡的过程：

复制代码

```
# 复制字节码到容器中
docker cp xdp-proxy.bpf.o lb:/

# 在容器中安装iproute2命令行工具
docker exec -it lb apk add iproute2 --update

# 在容器中挂载XDP程序到eth0网卡
docker exec -it lb ip link set dev eth0 xdpgeneric object xdp-proxy.bpf.o sec
```

除了使用命令行工具之外，你还可以在用户态程序中调用库函数来挂载 XDP 程序，从而避免引入额外的命令行工具依赖（比如，不再需要安装 iproute2 系统工具）。

libbpf 提供了一个 `bpf_set_link_xdp_fd(int ifindex, int fd, __u32 flags)` 函数，可用于把 XDP 程序挂载到网卡。这个函数需要网卡序号和 XDP 程序文件描述符作为参数，查询这些参数并挂载 XDP 的过程如下所示：

复制代码

```
    unsigned int ifindex = if_nametoindex("eth0");
    int prog_id = bpf_program__fd(obj->progs.xdp_proxy);
    err = bpf_set_link_xdp_fd(ifindex, prog_id, XDP_FLAGS_UPDATE_IF_NOEXIST|XD
```

这段代码中，挂载参数标志 `XDP_FLAGS_SKB_MODE` 等同于 `ip link` 命令中的 `xdpgeneric`，表示以通用模式挂载。

把上述代码保存到 `xdp-proxy.c` 文件中（你还可以在 GitHub 中找到完整的代码），再执行下面的编译命令，就可以将其编译为静态链接的可执行文件。采用静态链接的一个好处是容易在容器中分发，只需要把最终的二进制文件放入容器中即可运行，不再需要安装额外的依赖环境。

复制代码

```
clang -g -O2 -Wall -I. -c xdp-proxy.c -o xdp-proxy.o
clang -Wall -O2 -g xdp-proxy.o -static -lbpf -lelf -lz -o xdp-proxy
```

到这里，完整的 eBPF 程序就开发好了。它是不是可以正常工作呢？如果可以正常工作，性能又会怎么样？接下来，我们把它放到容器中测试一下看看。

## 性能测试

在终端中执行下面的 docker 命令，把 XDP 程序复制到负载均衡容器中，并执行 XDP 程序：

复制代码

```
# 复制XDP程序到容器
docker cp xdp-proxy lb:/

# 在容器中加载XDP程序
docker exec -it lb /xdp-proxy
```

然后，进入客户端容器终端中，执行 apk 命令安装 curl 和 wrk 工具，接着再使用 curl 访问负载均衡器：

复制代码

```
docker exec -it client sh

# (以下命令运行在client容器中)
/ # curl "http://172.17.0.5"
```

如果你看到 Hostname: http1 或者 Hostname: http12 的输出，说明 XDP 已经成功运行，并且它的负载均衡功能也是正常的。

接下来，继续在客户端容器终端中执行 wrk 命令，给负载均衡器做个性能测试：

复制代码

```
# (以下命令运行在client容器中)

# 安装wrk工具
/ # apk add curl wrk --update

# 执行性能测试
/ # wrk -c100 "http://172.17.0.5"
```

稍等一会，你会看到如下的输出（在你的环境下可能看到不同数值，具体的性能指标取决于运行环境和配置）：

复制代码

```
Running 10s test @ http://172.17.0.5
  2 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     6.37ms   11.17ms 295.43ms   98.97%
    Req/Sec     9.09k   422.06    10.09k    75.00%
  180889 requests in 10.02s, 31.39MB read
Requests/sec:  18048.65
Transfer/sec:      3.13MB
```

从输出中你可以看到，平均每秒请求数是 18048，每个线程的平均延迟是 6.37ms。回顾一下 12 讲 中的套接字 eBPF 程序的性能测试结果，它的平均每秒请求数是 15300，而每个线程的平均延迟是 6.88ms。这说明，相比套接字程序，XDP 程序在平均每秒请求数上提升了 18%。

最后，不要忘记清理今天的案例环境。由于所有服务都运行在容器中，我们只需要执行下面的命令，删除今天创建的所有容器，即可完成清理工作：

复制代码

```
docker rm -f lb client http1 http2
```

## 小结

今天，我带你使用 libbpf 开发了一个基于 XDP 的负载均衡服务。

XDP 程序在网络驱动程序刚刚收到数据包的时候触发执行。由于还未分配内核 SKB 数据结构，XDP 程序只能根据 TCP/IP 协议的封包格式，从原始网络包中提取所需协议层的数据，进而再按照需要进行改写或转发。XDP 程序修改过的数据包可以转发给相同或不同的网卡，再交给内核协议栈继续处理；或者，跟处理逻辑无关的包不做任何处理，直接交给内核协议栈进行处理。

今天的案例把 XDP 程序挂载到了容器的虚拟网卡上，由于采用了通用模式挂载，它的执行过程其实还是在内核中运行的。在实际生产环境中，你可以以原生模式或卸载模式把 XDP 程序挂载到支持 XDP 的网卡上，从而获得最优的网络性能。

## 思考题

最后，我想邀请你来聊一聊：

1. 在今天的案例中，我们把 XDP 程序挂载到了负载均衡容器中的网卡上。如果把它直接挂载到主机的 eth0 网卡，会有什么样的现象？
2. 针对今天的负载均衡场景，还有哪些方法可以进一步优化 XDP 程序的性能？

期待你在留言区和我讨论，也欢迎把这节课分享给你的同事、朋友。让我们一起在实战中演练，在交流中进步。

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独购买本课程，你将得 20 元

生成海报并分享

赞 4 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 12 | 高性能网络实战（上）：如何开发一个负载均衡器？

下一篇 阶段总结 | 实用 eBPF 工具及最新开源项目总结

## 精选留言 (2)

写留言

**莫名**

2022-02-15

1. 挂在宿主机的 eth0 网卡，会导致宿主机网络受影响，最直接的影响是来自外部的 ssh 连接异常，之后再执行 ssh 也登不进宿主机。（原因是源 IP 不是 CLIENT_IP 时将目的 IP 直接修改为 CLIENT_IP，源 IP 无论如何均修改为 LOADBALANCER_IP，进来的数据包无法正常被接收与响应）

...

展开

作者回复: 非常赞的答案👍 如果能把这些思路再都实现了就更好了😊

1

**codejw**

2022-02-16

老师，如果有多个xdp程序如何挂载呢，我有多个.o都挂载到xdp

作者回复: 如果是不同的网卡，多次调用挂载和加载的函数或者用 ip link set 命令都可以。但如果想挂载多个XDP程序到相同的网卡，那就需要5.10新增的freplace类型，具体细节可以参考 https://lpc.events/event/7/contributions/671/attachments/561/992/xdp-multiprog.pdf